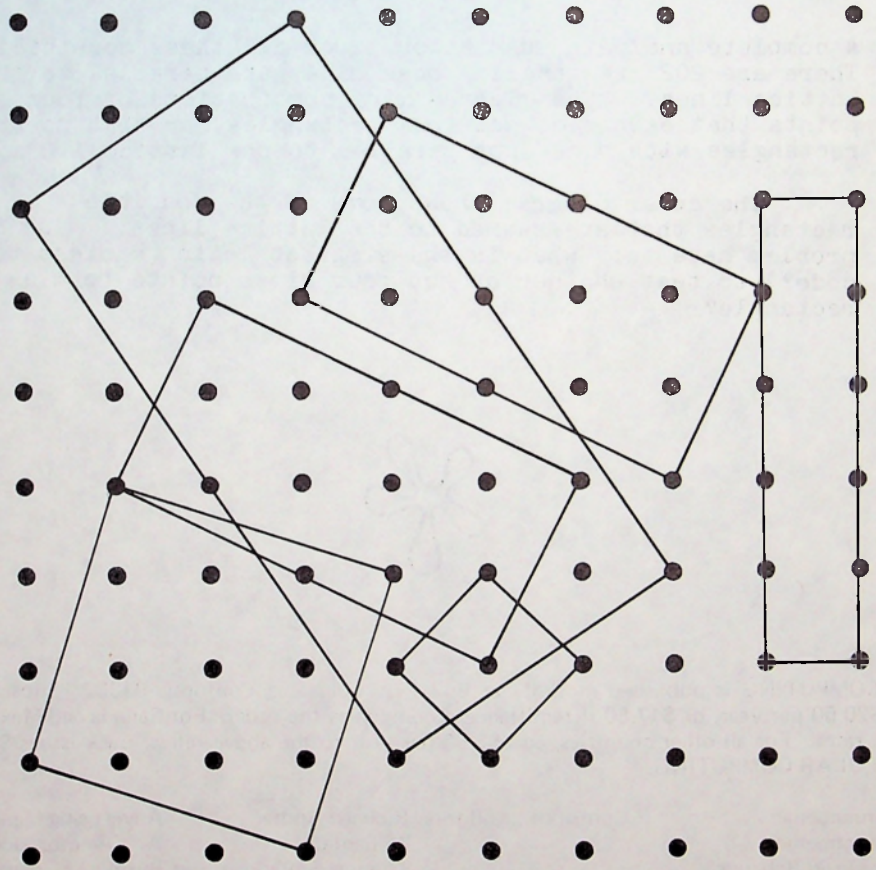
$2.50

50

Popular Computing

May 1977 Volume 5 Number 5



Rectangles On A Lattice

RECTANGLES ON A LATTICE

Given a 10 x 10 array of lattice points, as shown. By connecting four of the points, how many rectangles can be formed on the lattice?

The number of ways that 4 points can be selected out of 100 points is

$$_{100}C_4 = 3921225$$

A complete analysis must account for all these possibilities. There are 2025 rectangles whose sides are parallel to the lattice lines. That leaves many combinations of four points that either do not form rectangles, or that do form rectangles with sides not parallel to the lattice lines.

The cover diagram shows some of the possible rectangles that are skewed to the lattice lines. The major problem here is: what is the simplest logic (easiest to code?) to test whether or not four given points form a rectangle?

POPULAR COMPUTING is published monthly at Box 272, Calabasas, California 91302. Subscription rate in the United States is $20.50 per year, or $17.50 if remittance accompanies the order. For Canada and Mexico, add $1.50 per year to the above rates. For all other countries, add $3.50 per year to the above rates. Back issues $2.50 each.

QUADRATICS

In the expression:

$$21x^2 + Bx - 10 \qquad (P)$$

there are eight values for B that will make the expression factorable into rational integral factors; namely,

-1, +1, -29, +29, -11, +11, -209, and +209.

For this expression:

$$60x^2 + Bx - 60 \qquad (Q)$$

there are 46 possible values for B. In expression (P), the given coefficients each have just two factorizations, which combine to form the eight values for B, and all eight are distinct.

In expression (Q), on the other hand, each of the coefficients factors in six ways, which combine to give 72 (that is, six with six, but taking both the positive and negative values) choices for B, but with many duplicates.

The problem, then, is this. For the expression:

$$Ax^2 + Bx - C,$$

given A and C, how can one calculate the possible distinct values of B without running through all the possibilities and eliminating the duplicates?

Log 50 1.69897000433601880478626110527550697323181011853789 1

ln 50 3.91202300542814605861875078791055184712670284289729 1

$\sqrt{50}$ 7.07106781186547524400844362104849039284835937688474 0

$\sqrt[3]{50}$ 3.68403149864038660577982283357980722191725811743818 3

$\sqrt[10]{50}$ 1.47875763662831378554002296390226439875570578193461 3

$\sqrt[100]{50}$ 1.03989550282722797370118462810690234624216739261779 9

e^{50} 518470552858707246408 7.45332293348538482746910058384640190405693380685688479379539 8

π^{50} 72026719447158033063646 52.67235142347541936179524063386722193251713658004775995 1

$\tan^{-1}$ 50 1.55079899282174608617056849473815495414935150100104 4

Comments On "BASIC"

COMPARISON OF BASIC AND FORTRAN

John Maniotes James S. Quasney
Purdue University Calumet Campus

The article by J. L. Boettler in issue No. 46 was, in our opinion, an unfortunate comparison of Extended (STRONG) BASIC against the 1966 Standard (WEAK) Fortran. Naturally, when one tries to compare two programming langauages in this manner, it is like comparing apples against oranges.

We do not disagree with Boettler's conclusions regarding the ease of reading, writing, and using BASIC programs over Fortran programs. We do, however, disagree with the examples and arguments used to "prove" these conclusions, since these arguments may have left some of the readers confused or in doubt regarding the merits of Fortran.

If the article had compared a STRONG BASIC against a STRONG FORTRAN, then many of the weaknesses attributed to Fortran on page 16 would not exist. In addition, if the article had fixed the operating environment (that is, interactive or batch) for both languages or had permitted non-standard extensions to Fortran, then again many of the weaknesses of Fortran would have been non-existent.

Let's consider Boettler's first program to print the product of 5 and 7. Furthermore, let us permit this problem to be done in an interactive environment for both languages:

BASIC (Interactive)	Fortran (Interactive)
10 PRINT 5*7	K = 5*7 PRINT, K

By using an interactive environment and the non-standard "free format" capability for Fortran, many of the questions raised on page 15 disappear. Furthermore, since some interactive Fortran systems do have a non-standard "desk calculator mode," we can write the following: PRINT, 5*7 which is equivalent to the BASIC version. With regard to Boettler's problems 2 and 3, similar in-depth arguments can be made.

Another problem with Boettler's article dealt with the phrases "a standard Fortran" and "a strong BASIC language." By "a standard Fortran," does Boettler refer to the ANSI Standard Basic Fortran (which is a subset of the ANSI Standard Fortran) or to what? Since these phrases were not defined precisely, we have difficulty assessing the meaning behind each phrase.

Currently, there are a variety of non-standard dialects, levels, or versions of BASIC, some of which are called "Tiny-BASIC," "Full BASIC," "Extended BASIC," "Super BASIC," etc. Currently, there is no official ANSI Standard BASIC, although one is in the process of being issued.

As far as Fortran is concerned, the only official ANSI standards are the two 1966 versions which are woefully out of date with current Fortran systems. However, since 1966 many non-standard extension have been implemented into Fortran systems on many different kinds of computers, and some of these extensions have been incorporated into the ANSI 197X Standard.

One gets the impression from reading Boettler's article that not much progress has been made in developing "STRONG FORTRAN" systems and that all Fortran systems are limited and restrictive. On the contrary, many Fortran systems permit many of the following operations that Boettler defined as weaknesses. For example, many Fortran systems permit the use of:

1. PRINT instead of WRITE statements
2. Free format
3. Multiple statements per line
4. String functions
5. Real subscripts (which later are truncated to integer form)
6. Many forms of subscripts (not just 7)
7. Subscripted subscripts
8. Do-"While" loops instead of Do-"Until" loops
9. Mathematical relational operator symbols instead of abbreviations
10. IF-THEN-ELSE statements (Now in the 197X ANSI Standard)

even though the 1966 ANSI standards do not.

In order to put BASIC into proper perspective, the weaknesses of BASIC should also be highlighted. For example, many dialects of BASIC (including the proposed ANSI Standard at this stage of development):

1. Have a restrictive character set, thereby making meaningful variable names and meaningful function names difficult to devise.
2. Encourage GO TO's because of the lack of block structure in BASIC.
3. Encourage the use of weak constructs such as IF-THEN instead of IF-THEN-ELSE. Hence, long BASIC programs tend to have a spaghetti-like structure.
4. Have primitive subroutine capabilities which do not allow local variables to be declared. In addition, no parameter passing mechanism exists as in other high level languages.
5. Lack efficient statements to open, close, read, write, sort, and merge files.
6. Have a restricted set of one or two data types.

Finally, if one compares a "STRONG BASIC" used in an interactive environment against a "WEAK FORTRAN" used in a batch environment, then it is no wonder that BASIC is so superb. However, if one compares a "STRONG BASIC" against a "STRONG FORTRAN" in which both are used in an interactive environment, then which language is the "best" becomes debatable as far as the user is concerned.

□

RECIPROCAL SEQUENCE

The sequence defined by:

$$A_n = \frac{1}{A_{n-1}} + \frac{1}{A_{n-2}}$$

(with $A_1 = 123$ and $A_2 = 56$) begins:

.025987
38.498304
38.506422
.051945
19.277151
19.303056
.....

Without leaping to your computer or pocket calculator, determine the 1000th term of the sequence.

PROBLEM 173

Vassilakis

During the years that Software Age was being published, George Vassilakis ran a monthly column called "X Tran's Adventures in Fortran." Usually, he presented a short Fortran program, with questions about its action during compilation or execution. His classic problem bears repeating:

```
      DØ 7 I = 1, 10
      I = I + 1
      WRITE (6,5) I
5     FØRMAT (I5)
7     CØNTINUE
      STØP
```

The program clearly violates a rule; namely, Don't operate on the index of a loop within the loop. As is customary with reference manuals, the Fortran manuals say "don't do that" without saying what happens if you do. Anyone who uses a high level language should be aware of what will happen to him when he violates a rule. Moreover, for such a clear-cut case, one might expect the compiler to note the error with an unambiguous error message and have execution of his program inhibited.

At the time Mr. Vassilakis presented this problem (when 360's and Sigma series machines were new, and 7094's were still around), 90% of the Fortran's tested compiled the program with no error message, and printed one of the following:

```
2, 4, 6, 8, 10
2, 3, 4, 5, 6, 7, 8, 9, 10
1, 3, 5, 7, 9
```

Some Fortran's gave an error message, but then compiled and executed anyway. A time-sharing system gave this message:

```
ERROR 1046 AT STATEMENT 2
```

Considered by itself, this is amusing. If the user can find the book that lists all the error messages, he will probably find that it ends with message 1045. In this case, however, the message continues:

```
THE RUNNING INDEX IN A DO MAY BE CHANGED WITHIN THE LOOP
```

which is a fine example of the mis-use of English.

We would like to see the results of running Mr. Vassilakis' program with today's Fortran's, to see if the world has advanced in any way.

A SUM OF A DIFFERENT KIND

Suppose we sum an infinite series in a different way. Consider the series:

$$\frac{1}{1} + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \frac{1}{11} + \frac{1}{13} + \frac{1}{17} + \cdots + \frac{1}{k} + \cdots$$

where the denominators are successive odd primes. We will sum the terms in natural order, but hold the sum to less than 2; that is, we will reject any prime, k, for which adding 1/k would put the sum over 2. The accompanying table shows the calculation through k = 23. Any value of k through k = 957 will carry the sum over 2.

k	1/k	sum
1	1.00000000000000	1.00000000000000
3	.33333333333333	1.33333333333333
5	.20000000000000	1.53333333333333
7	.14285714285714	1.67619047619047
11	.09090909090909	1.76709956709956
13	.07692307692307	1.84402264402264
17	.05882352941176	1.90284617343440
19	.05263157894736	1.95547775238176
23	.04347826086957	1.99895601325133

Thus, the next fraction to be added must have a denominator greater than 957, and the addition of that fraction will push the sum closer to 2.

So the problem is this: What are the next values of k in this summation?

ART OF COMPUTING 13 FJG

Where to Begin?

In this article, we propose to show the analysis of a simulation problem, up to the point of actual coding, taking into consideration such things as:

- ★ The choice of an algorithm.
- ★ The maintainability of the resulting program (and its readability, and its comprehensibility)
- ★ The sensible tradeoffs between efficiency of compilation and execution
- ★ The proper utilization of programmer time.

The problem is this: we wish to simulate a card game (e.g., Blackjack) and must begin with a scheme for shuffling a 52-card deck. For simplicity, we will consider this deck to be the numbers from 1 to 52 (and where necessary to show examples, we will use the numbers from 1 to 20).

The problem, in technical terms, is thus that of producing random permutations. Within a much larger problem--the card game itself--we need a subroutine that will output on demand a fresh arrangment of the numbers from 1 to 52. We will assume the availability of another subroutine--a random number generator (RNG)--that will produce on demand random integers, one at a time, in any desired range. Much of what follows hinges on the quality of that generator; see the article on "Random Numbers"--Number 5 in this series--in issue No. 21.

If the problem is now clearly in mind, this is the time to plan ahead for the desirable criteria:

1. Do we want a high degree of randomness on each call of the subroutine?

2. Is speed of execution for the subroutine something to worry about?

3. Will we eventually want 10 random permutations, or 1000, or a million? That is to say, having decided the first two questions, is ease of programming a criterion, or will a quick and dirty algorithm suffice?

There are $8.0658175 \cdot 10^{67}$ possible permutations of the 52 numbers. By "a high degree of randomness" it is meant that each of these permutations has the same chance of occurring as any other.

Let us now consider various algorithms that will accomplish our task.

1. Create a block of 104 words, addressed at B, B+1,...,B+103. In words B+1, B+3,...,B+103 generate the numbers from 1 to 52. In words B, B+2,...,B+102 generate random numbers in the range 1 to 100,000. Sort the 52 word pairs, using the left hand word of each pair as the sorting key. When the 52 pairs are in ascending order on the random numbers, the 52 right hand words constitute the desired output. This method is positive and definite and will yield a high degree of randomness for the 52 numbers.

2. Clear a block of 52 words to zero. Call the random number subroutine to produce one number in the range 1-52. If that number has not previously been called, enter it into the block. If it has been previously called (which means checking against a list of the 52 numbers in another block), disregard it. When all 52 numbers have been entered into the block, then the contents of the block constitute the required output.

This method also has a high quality yield, but its speed is variable. It will produce the first 40 numbers or so of the required output rapidly, but then start to slow down drastically for the remainder. The last few will take more time than all the rest.

This procedure can be speeded up. Suppose that it is allowed to run to the point where 48 numbers have been selected, and that the numbers yet to be selected are 5, 11, 37, and 48. There are 24 ways in which these four numbers can be permuted:

05	11	37	48	37	05	11	48
05	11	48	37	37	05	48	11
05	37	11	48	37	11	05	48
05	37	48	11	37	11	48	05
05	48	11	37	37	48	05	11
05	48	37	11	37	48	11	05
11	05	37	48	48	05	11	37
11	05	48	37	48	05	37	11
11	37	05	48	48	11	05	37
11	37	48	05	48	11	37	05
11	48	05	37	48	37	05	11
11	48	37	05	48	37	11	05

and one of the 24 permutations can be selected by calling for a random number in the range 1-24. In similar fashion, the speedup process could be applied to the last 5 numbers (with now the choice of one out of 120 possible permutations to be selected at random),or even to the last 6 numbers.

3. Generate the numbers from 1-52 in a block of words addressed at T, T+1,...,T+51. Let the random number generator generate addresses in the range from T to T+51, and generate one such address. The contents at that address is moved to the output array, and a zero replaces it in block T. If the selection process arrives at a word whose contents is zero, that selection is ignored. This process could proceed in that way, but would suffer the same slowdown as in method 2 when the last few numbers have yet to be moved. A better scheme would be to shorten block T by one word after each selection, and correspondingly shorten the range of addresses sampled by the RNG.

4. A physical deck of cards is sometimes manually shuffled by separating the deck into two parts (usually of unequal size) and <u>interfiling</u> the two parts. Randomness is injected into this process by having one's fingers allow two or three cards from each part to merge together at random times. This process itself could be simulated. Generate the numbers from 1-52. Use the RNG to select the size of one part (say, a number in the range 22-30). Then merge the two parts, using the RNG to choose how many numbers (say, in the range 1-3) should pass from each part. The merged set of numbers is then subjected to the same procedure, as many times as is necessary to scramble the deck. The merge operates on addresses, rather than on their contents.

5. Generate the numbers 1-52 in a block T that is regarded as cyclic; that is, following words T+50, T+51, come words T, T+1, T+2, etc. The block T is then taken as a rotating roulette wheel. The RNG dictates the amount of rotation between successive selections of the numbers. As each number is selected and moved to the output array, its word is cleared to zero. As in method 3, the wheel can be shortened by one word after each selection.

6. Many methods exist to generate systematic permutations. Using one of these methods, the RNG can be used to indicate how many permutations should be by-passed for each one required. This method is included only for completeness; it does not seem to be practical. (Then again, in the right hands, it might turn out to be the most practical method of all.)

7. For many decades, the function of cipher devices has been essentially to generate random permutations. Thus, the logic of such devices could be simulated for the task at hand. In particular, the rotor mechanisms could be programmed on a computer. Again, this method is included only for completeness. For details of rotor cipher devices, see David Kahn's The Codebreakers.*

8. Create a block of 52 words, initially containing zeros. Use the RNG to generate random addresses in the block, and fill them with the numbers from 1-52 in succession until the block is filled. This method differs only slightly from methods 2 and 3.

9. There is another way in which cards are mechanically shuffled. Suppose we have given a block of 52 words (T through T+51). We use the RNG to select three numbers:

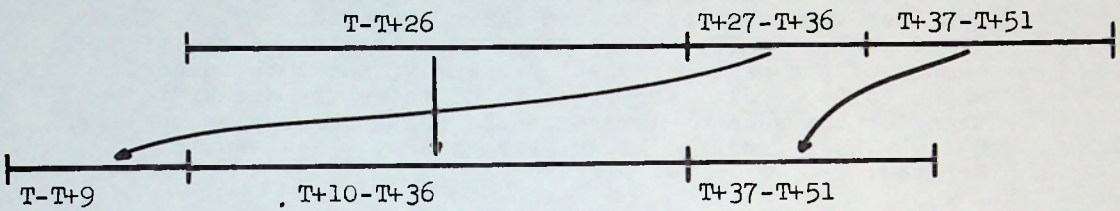
an address in the range T+1 through T+40

a number in the range 1-30, chosen such that this number added to the address chosen above does not exceed T+51

a number in the range 1-2 (1 for "left," 2 for "right.")

*Macmillan, 1967.

These three numbers dictate the portion of the "deck" to be shuffled. For example, if the three numbers are T+27, 10, and 1, it can indicate that words T+27 through T+36 are to be moved to the left:



with the new array condensed back into the original space. As with any mechanical shuffling process, the procedure is repeated enough times to insure thorough scrambling. This method will not yield good randomness, but could be useful as an "expander"; that is, a method to use for a few times between applications of other, more time-consuming methods.

10. Given a useful permutation, another scheme for expansion would be to add a constant to each number and reduce modulo 52. Suppose we had the permutation (on a range of 20) shown in the first column of the accompanying figure. A constant is added to each number, as shown, to produce a new and seemingly fresh arrangement.

Example for method 10

old permutation		result	reduced result
14		29	10
03		18	19
13		28	09
16		31	12
18		33	14
04		19	20
17		32	13
20		35	16
12		27	08
08		23	04
01		16	17
19		34	15
15		30	11
06		21	02
05		20	01
11		26	07
02		17	18
10		25	06
07		22	03
09		24	05

and we add 15 to each number and reduce modulo 20 plus one.

11. Another expansion method is that of autocoding. Consider the last permutation shown for method 10. Its first number is 10, which can be used to dictate the first number of the next permutation; namely, the 10th number, which is 04. Similarly, the 19 dictates the number to be used for the second element of the next permutation; namely, 03. Thus, from the given arrangement, a new one is derived:

04 03 08 15 02 05 11 07 16 12 18 01 17 19 10 13 06 20 09 14

The output from this method is about as non-random as can be, being directly related to the most recent output. However, for the purpose of the larger program which it serves, this scheme might be quite acceptable for short runs.

12. Still another possibility is to use the sum of two successive permutations to derive a third. For example, suppose we have permutations A and B from any source:

A 09 10 02 18 16 03 13 05 06 19 15 01 20 12 04 07 14 08 11 17

B 04 12 20 15 19 06 05 17 11 08 14 07 09 01 13 10 02 03 16 18

C is formed by adding A and B, term by term:

C 23 22 22 33 35 09 18 22 17 27 29 08 29 13 17 17 16 11 27 35

and reducing, as before, modulo 20 plus 1:

C' 04 03 03 14 16 10 19 03 18 08 10 09 10 14 18 18 17 12 08 16

This leads to missing numbers (9 of them in this case) and many duplicates. The duplicates have to be replaced by the missing numbers, and the process gets messy.

All of this analysis indicates that the sub-problem is not simple or obvious. Some of the randomizing methods can be discarded immediately as either impractical or requiring excessive work in coding (not to overlook the chores of debugging and testing). Of those remaining (methods 1, 2, and 3 are probably the best), the choice must be made that balances the programming effort against machine efficiency.

Suppose we choose method 1. There is immediately a new problem; namely, how shall the sorting be done? Could we use bubble (interchange) sorting on 52 word-pairs? Bubble sorting will require, in the worst case, 1326 comparisons and interchanges for each permutation produced, or about 500 for the average case. Is that excessive? It certainly is, if we are going to need several thousand random permutations. On the other hand, an efficient sorting scheme (e.g., a Shell sort) is more difficult to code, debug, and test. If our game playing program will use only a few hundred shuffles, perhaps we'll get there faster and better if we do use a bubble sort. Go back to the first paragraph--those questions should be answered before flowcharting or coding begins.

A beginner is apt to plunge into the coding phase of his card game program without regard to the criteria raised in the first paragraph. The step "shuffle the deck" is seen as only a minor nuisance, to be cared for later. The time to care for those minor nuisances is at the start of a new project, not in the middle of it. □

Squares In Arithmetic Progression

PROBLEM 176

It is fairly easy to find three perfect squares that form an arithmetic progression, such as:

			common difference
7^2 = 49	17^2 = 289	23^2 = 529	240
28^2 = 784	52^2 = 2704	68^2 = 4624	1920
63^2 = 3969	117^2 = 13689	153^2 = 23409	9720
9406^2 = 88472836	14450^2 = 208802500	18142^2 = 329132164	120329664

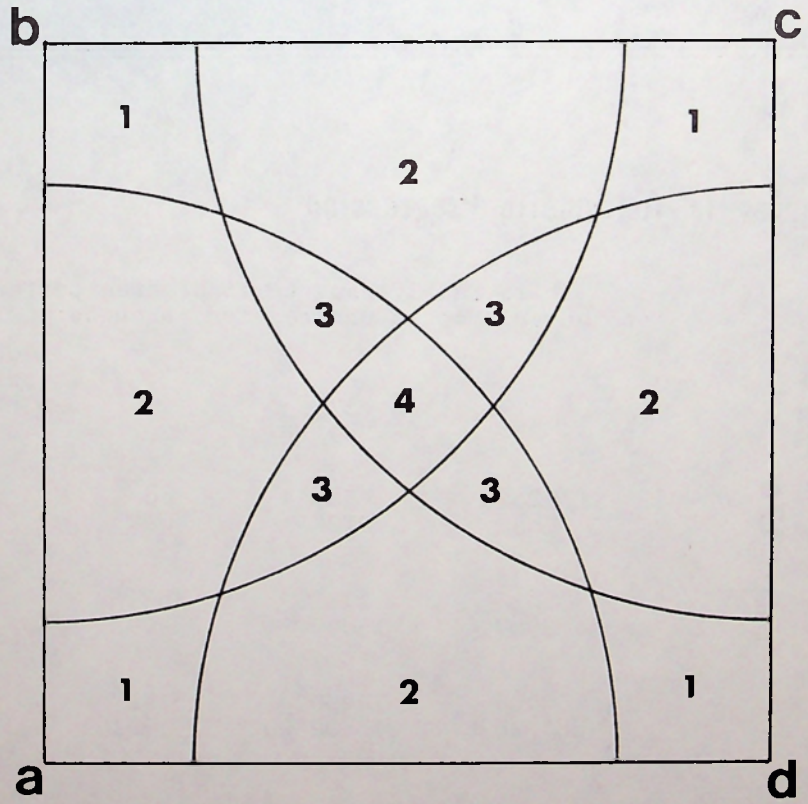
Can you find four perfect squares that are in arithmetic progression?

THE GOAT PROBLEM

A goat is tethered at each vertex of a square field that is 100 meters on a side. The tether for each goat is 80 meters long. Of the 10000 square meters of grass, it is assumed that each goat eats all of his exclusive area; 1/2 of that which he shares with one goat; 1/3 of that which he shares with two goats; and 1/4 of the grass common to all four goats. How much of the grass does each goat eat?

This is an old puzzle problem. Due to the symmetry of the situation, the answer is that each goat consumes 1/4 of the grass.

The Problem is: what are the various areas involved?



A FORMULA FOR GENERATING PRIME NUMBERS

R. W. Hamming

It is often carelessly said that there is no formula that generates only prime numbers. But consider the following.

Let G be the binary number .0110101000101...

The rule for forming G is that it has a 1 in the Kth position if K is a prime and otherwise it has a zero. Let the formula be:

$$K\left[2\left\{2^{K-1}G - \left[2^{K-1}G\right]\right\}\right] = \begin{cases} \text{a prime number} \\ \text{zero} \end{cases}$$

where the square brackets indicate the greatest integer in the number. Starting with the right end of the formula,

$$2^{K-1}G$$

shifts the binary point in front of the Kth digit. Then the curly brackets give the fractional part of

$$2^{K-1}G.$$

The next multiplication by 2 and the second square bracket isolates the Kth digit. The final multiplication by K produces the prime number, or else zero.

The formula immediately appears as a fraud; you need to know the answer before you start! The number G appears to be well defined mathematically. The operations in the formula are well defined. It is one of those existence theorems. Among all the uncountable number of binary numbers between zero and one, there is surely the number G, and the formula follows.

In the currently dominant philosophy of mathematics, it is a well-defined formula, but there is an active minority that denies that the number G exists. How do you feel about it? Does the number G exist?

KNIGHTS AWAY

On an N-sided chessboard, what is the maximum number of pieces that can be placed so that no two are closer than a knight's move from each other?

Possible patterns for the first 8 cases are shown on the facing page, leading to the table:

N	n	N	n
1	1	5	6
2	1	6	8
3	2	7	10
4	4	8	13

What are the values of n for N = 9, 10, 11,...?

PROBLEM **178**

Gosper

It is obvious that the subject of high precision arithmetic intrigues us. We keep getting references to the work of R. W. Gosper, currently at Stanford, as being outstanding in this area. So we wrote to Mr. Gosper to inquire into his work:

> "Can you take an arbitrary constant, say the 7th root of 306 [see PC48-6] and calculate a thousand places or so?"

Mr. Gosper could. We now have that constant to 2800 significant digits, beginning with:

2
2651817862 9569774774 5672132632 7241189213 1448486969
9946531670 9048414227 4205493475 1549122216 2148541898
4950568931 2038116592 0897070048 2412823318 8382049008
9862401932 3379113007 9912490219 2500532037 4618587150...

PC50-19

PC50-20

Inventory Clearance Sale

Limited Time Only

Complete Your File of *Popular Computing*

Most Back Issues Are Still Available:

On orders received up to June 30, 1977:

1. For 10 or more of the issues that are still available.
2. $1.00 (U.S.) per copy.
3. Chosen from issues 1 through 39 only.
4. Mailed to one address by surface mail.

JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC		
			1	2	3	4	5	~~X~~	7	~~X~~	9	Vol. 1	1973
10	11	12	~~X~~	14	~~X~~	16	17	~~X~~	19	20	21	Vol. 2	1974
22	23	24	25	26	27	28	29	30	31	32	33	Vol. 3	1975
~~X~~	35	36	37	38	39	40	41	42	43	44	45	Vol. 4	1976
46	47	48	(49)	[50]	51	52	53	54	55	56	57	Vol. 5	1977